Transaction Processing and Consistency Control of Replicated Copies during Failures in Distributed Databases

BHARAT BHARGAVA

BHARAT BHARGAVA is an Associate Professor of Computer Sciences at Purdue University. He received his Ph.D. degree from Purdue University in 1974. He joined the Department of Computer Sciences at Purdue University in 1984. His research involves concurrency control and reliability issues in distributed database systems and software fault-tolerance. He is at present implementing a robust and adaptable distributed database system called RAID. He is the founder of the IEEE Computer Society's Symposium on Reliability in Distributed Software and Database Systems, and he organized the IEEE workshop on Design Principles for Experimental Distributed Systems in October 1986.

Dr. Bhargava has published in *IEEE Transactions on Software Engineering, IEEE Transactions on Computers, Journal of Systems and Software, Information Science*, and other journals. He has edited a monograph *Concurrency Control and Reliability in Distributed Systems*, published by Van Nostrand Reinhold in 1987.

ABSTRACT: Consistency of replicated copies is difficult to maintain and recover during multiple failures of sites and network communication in a distributed database system. Transaction processing must continue as long as a single copy is available. But in a multiple failure environment, each operational site must make correct decisions about which copy to update and which one will be updated by the recovery system. This requires refreshing the copies on failed sites that missed the updates and doing this correctly while other transactions are updating and some more sites are either failing or recovering. This problem has been classified as the "replicated copy control problem." In this paper, we present several ideas that are necessary to attack and manage this problem. We introduce the ideas of session numbers, nominal session vectors, fail locks, and view serializability and discuss their role in transaction processing on operational, recovering, and partitioned sites. We have experimented with many of these ideas in a prototype system called RAID and we present the implementation issues. There is little overhead associated with our approach if no failures occur.

This research has been supported by a grant from Sperry Corporation and the National Aeronautics and Space Administration.

This paper was presented at the conference on Current Issues in Database Systems, Rutgers University, Newark, May 27, 1986. By permission.

KEY WORDS AND PHRASES: Replicated copy control problem, multiple site failures, consistency control of replicated copies, distributed database system.

1. Introduction

IN A DISTRIBUTED DATABASE SYSTEM, replicated copies of data are stored at different sites to increase availability. Due to a hardware/software crash, a site may fail and stop processing. Similarly due to the communications system failure, sites may partition into different groups that are unable to access data and exchange messages. The system is not able to control the event of failure or partition, but has the responsibility for continued transaction processing and maintenance of the replicated copies of the database in a consistent manner. These two tasks require special attention when a site is recovering and integrating back to the operational system. For transaction processing, three classes of transactions must be considered in such an environment:

(a) transactions that arrive and finish while the status of the up and down sites does not change in the meantime;

(b) transactions that arrive when a particular site is up and finish after this site has failed or network partition has occurred;

(c) transactions that arrive when a particular site is down/partitioned and finish when such a site is recovering or the network is merging.

For transactions of type (a), our research shows that the failed site can be ignored until its recovery. Transactions of type (b) should be aborted to avoid the overhead of additional rounds of messages if at each round a new failure is detected. Transactions of type (c) require either that the recovering site be treated as failed or else that careful coordination of the events at the operational and recovering sites be carried out for successful commitment. This research is an attempt to increase database availability, and our protocols allow transaction processing if even a single copy is available and also maintain the consistency of the database. A weighted voting approach was suggested in [7] where the read quorum and the write quorum must overlap by at least one site. This approach requires establishing the size of the quorum at the beginning of the transaction. Even though the transaction can succeed even if some sites fail, the sum of the read quorum and the write quorum must equal the initial quorum.

Although the read availability is increased by data replication, the write availability is decreased. For example, under the *strict write-to-all* scheme, write operations to the logical data items are interpreted as writing to all replicated physical copies. If one of the replicated copies is unavailable due to a site failure, the transaction is aborted (or blocked until the failed site is operational). An attempt to increase write availability can result in inconsistent data. Hence the site recovery problem plays a key role in replicated distributed database systems.

The use of multiple reliable *message spoolers* [8] is one practical solution to this problem. All update messages addressed to an unavailable site are saved reliably (in multiple spoolers), and the recovering site processes all its missed messages before

resuming its normal operations. This method is not suitable for systems in which sites may be down for a long time, and this solution does not take advantage of data replication. We use the spoolers only for a short duration at a critical time in the recovery process.

There are few solutions to identify out-of-date copies at the recovering site. It is possible to renovate data items at the recovering site by utilizing up-to-date copies available at other sites rather than redoing all missed operations. It is not a good idea to copy the whole database from an operational site to the recovering site. An approach was proposed by [1] where, briefly, the copies at the new site are brought up to date after they are written by user transactions or by *copier transactions* that get the data from copies at operational sites. The solution is based on the fact that a distributed database system must run a correct concurrency control algorithm and relies on the concurrency control algorithm to handle all synchronization for recovery and transaction processing. This solution is not suitable for the site recovery problem if sites are up and down dynamically and if one likes to isolate the concurrency control problems from the recovery problem. Below we show two problems that can occur if care is not taken during site recovery.

1.1. Examples of Replicated Copy Control Problems

A recovery algorithm depends on the computational model, especially the semantics of logical operations in the presence of failures. Without a clear specification of the computational model, a recovery algorithm cannot be proven to be correct. For example, if a logical read operation is interpreted as reading from any available copy, and a logical write operation as writing to all currently available copies, even when failure or recovery are in progress, the following scenario may result in an inconsistent status of the database and the recovery performed by the algorithm given in [1] will have problems.

Example 1: Incorrect History

Transaction T_a reads X and writes Y; transaction T_b reads Y and writes X. Both X and Y have two copies at Site 1 and Site 2, called x_1, x_2 for (X) and y_1, y_2 for (Y), respectively. A copier transaction reads an up-to-date copy and writes the copy on the recovery site. A history

$$R_a[x_1]R_b[y_1] \text{ (site 1 crashes) } W_a[y_2]W_b[x_2]$$

can be accepted by a concurrency control protocol. Since both T_a and T_b have written to all currently available copies, they can assume that they are done. When Site 1 recovers, x_1 and y_1 may be updated by copier transactions T_c and T_d and the history continues as follows:

$$\text{(site 1 recovers) } R_c[x_2]W_c[x_1]R_d[y_2]W_d[y_1]$$

The copier transaction T_c has the same effect as if T_b writes to x_1, and T_d has the effect as if T_a writes to y_1. Therefore, the effect of the above history is equivalent to the effect of the history

$$R_a[x_1]R_b[y_1]W_a[y_2]W_b[x_2]W_b[x_1]W_a[y_1]$$

Unfortunately this is a nonserializable history.

Example 2: Missing Update

Let T_a and T_b be as in Example 1. Let logical items X and Y have copies at both Site 1 and Site 2 as before.

Site 2 crashes and then recovers during the history execution. The copier transaction T_c reads the copy x_1 at Site 1 and writes the copy x_2 at Site 2.

A history as shown as follows is allowed to occur.

$$H = \text{(site 2 crashes) } r_a[x_1]r_b[y_1]w_a[x_1] \text{ (site 2 recovers) } r_c[x_1]w_c[x_2]w_b[x_1]$$

The copy x_2 reflects the update due to $W_a[X]$, but the update $W_b[X]$ is lost, whereas the copy on Site 1 reflects the updates due to both $W_a[X]$ and $W_b[X]$. This will cause the two copies to be inconsistent.

We classify the problems discussed above as falling under the subject of *replicated copy control*. When there are no failures, replicated copy problems can be handled by distributed concurrency control algorithms [11]. Since we consider replicated copy control only when site failures and network partitions occur, this is a substantially new research topic. The problems under study here are different from the stable storage and crash recovery problems which deal with ensuring that updates of a committed transaction are flushed correctly on the disk or the transaction is rolled back using logs and audit trails and other means. The protocols used for processing transactions during a network partition [6] assume the detection of a partition, and all read and write actions must occur while the partitions exist. Our research ideas can take into account the problems that occur when a partition occurs or merges during the life of a particular transaction. Finally, replicated copy control is different from the commit and termination protocols [12, 13] although it works in conjunction with them. Such protocols deal with the correct commitment and abortion of a transaction rather than the consistent updating of the replicated copies. Since a site upon recovery must commit or abort the transactions which were being processed when the site crashed, consistent with the decisions made by operational sites, the goal of the commit/termination protocols is to ensure that the recovering site makes consistent decisions. We deal with the events at the recovering site while the operational sites do very little for the recovery protocol and process transactions as under normal conditions.

1.2. Focus of the Paper

In this paper, three ideas are introduced. These ideas are useful for transaction

processing when failures of site or network communication occur and provide solutions for recovering the consistency of the database at the failed site. Together they give insight and a protocol for replicated copy control.

(a) Protocol Design for Site Recovery Problems

A protocol for site recovery problems has been presented in [5] where details and proofs of correctness are given. Here we will outline the basic ideas. These ideas help in discussing and clarifying the fail lock concept.

(b) Fail Locks

Fail locks are used to identify and update out-of-date copies. The fail locks are set when an update is made to a copy while another copy is unavailable. They represent the notion that update to a certain copy has failed.

(c) View Serializability and Network Partitioning

View serializability is an extension of the serializability theory for concurrency control. View serializability [14] allows correct views to the read-only transaction even if this view may not be most current. We will use this notion for the continued processing of the read-only transactions during network partitioning.

2. Protocol Design for Site Recovery Problems

SITE RECOVERY IS THE PROBLEM of integrating a site into a distributed database system (DDBS) when the site restarts after a failure. There are two different problems and approaches under the term "site recovery" in the literature. The first concerns the resolution of transactions at all the sites. That is, upon recovery, the failed site should commit or abort the transactions that were being processed when the failure occurred, consistent with the decision made by other operational sites in the system. Commit, termination, and recovery protocols make it possible for the recovering site to make correct decisions on these transactions [13].

In this paper, first we briefly present the ideas that allow normal transaction processing at the operational sites and next address the second problem in depth, which deals with the consistent recovery of the database. In our approach the failed/recovering site is relieved about ongoing transactions except that it should get a consistent copy of the database before any transaction processing. When a failed site recovers and integrates with the system, the consistency of the entire database is threatened because the data items at the recovering site may have missed some updates. The recovery algorithm should enable the recovering site to start processing transactions as soon as possible, which requires update copies of all data items. Research ideas are further discussed in section 3 under fail locks. The recovery procedure allows the recovering site to resume its normal operations on certain data

items immediately and on all others via incoming updates on the operational sites or by demanding a current copy as the need arises.

The basic idea used for correct transaction execution at the operational sites is to maintain a consistent view of the status (up or down) of all sites. The view need not be the exact status of other sites, but is the status as perceived at each site. The *session number* is used to represent the state of a site, while the *nominal session number* is used for the session number as perceived by other sites. The nominal session numbers are maintained consistently by *control transactions*, which run concurrently with user transactions.

2.1 Background

In this paper, the users' view of an object is called a *logical data item*, or a *data item*, denoted X. A data item is stored in the DDBS as a set of *physical copies* or *copies*. The copy of X stored at site k is denoted x_k, and the fact that X has a copy at site k is denoted $x_k \in X$. The users manipulate the database via transactions. To process a transaction on multiple copies of data items, the strict *read-one/write-all* (ROWA) strategy can be described as

$$\text{READ}(X) = \vee \{\text{read}(x_k),\ x_k \in X\},$$

$$\text{WRITE}(X) = \wedge \{\text{write}(x_k),\ x_k \in X\},$$

where $OP = \vee \{op\}$ means that OP is interpreted as at least one of the op's, and OP fails if no op succeeds; and $OP = \wedge \{op\}$ means that OP is interpreted as all the op's and OP fails if any one of the op's fails.

In a system using the strict ROWA scheme, all other copies become available for update and hence site failures never result in inconsistent data. Consequently, site recovery (in the sense of the consistent recovery of the database) is unnecessary. However, the degraded availability for write operations makes the strict ROWA scheme impractical. An alternative to this scheme is the *read-one/write-all-available* (ROWAA) protocol. Intuitively, if a transaction knows that site k is down, it should not try to read a copy from site k, or send an update to site k. ROWAA not only saves the time otherwise wasted because of waiting for responses from an unavailable site, but also reduces the possibility of aborting or blocking transactions.

2.2. Session Numbers and Nominal Session Numbers

As far as recovery is concerned, a site has three distinguishable states. We say a site is *down* if no DDBS activity is going on at the site. A site is *recovering* if it is in an early stage of its recovery procedure. The site is *operational* or, simply, *up* if all database copies are available for reading and writing. In some situations a site may

be considered up and transaction processing may be allowed on some identifiable consistent data items while other copies are being renovated. We say a site is *not operational*, when the site is either down or cannot do any transaction processing.

An *operational session* of a site is a time period in which the site is up. Each operational session of a site is designated with an integer, *session number*, which is unique in the site's history, but not necessarily unique systemwide. If a site is not in an operational session, its session number is undefined. For simplicity of description, however, we say that the site has session number 0 if it is not operational (assuming 0 is never used as a session number for an operational session). The session number of site k is denoted as $as[k]$.

Because sites are up and down dynamically, it is not always possible for a site to have precise knowledge about the session number of another site. In order to have a consistent view of the session number of a particular site i in the system, we augment the database with additional data items, called *nominal session numbers*. We use the notation $NS[k]$ for the data item indicating the nominal session number of site k, and NS for the vector composing $NS[1], \ldots, NS[n]$. Note that the nominal session number of the site k may differ from the actual session number $as[k]$, but the difference should be kept tolerable as far as possible. The copy of $NS[k]$ at site i is denoted $ns_i[k]$. In the ROWAA scheme, if a transaction, initiated at site i, reads the nominal session vector ns_i, its logical operations are interpreted by the site i as:

$$\text{READ}(X) = \vee \{\text{read}(x_k),\ x_k \in X \text{ and } ns_i[k] \neq 0\},$$

$$\text{WRITE}(X) = \wedge \{\text{write}(x_k),\ x_k \in X \text{ and } ns_i[k] \neq 0\}.$$

Each request for reading or writing a physical copy at site k carries $ns_i[k]$, the session number of site k perceived by the transaction. The site k first checks this number against its actual session number, $as[k]$. If they are not equal, the request is rejected or, during recovery, it can be placed in a buffer. Otherwise, the request is carried out.

2.3. Copier Transactions

A copier transaction is responsible for refreshing a particular unreadable data copy. It reads (a copy of) the nominal session number, locates a readable copy (with the help of fail locks), and uses its content to renovate the unreadable copy. A copier transaction is run at the recovering site whenever a request for an unreadable data item is made. The values of the unreadable data items are inconsistent with the values of the copies on the operational sites.

2.4. Control Transactions

The transition of nominal session numbers is done by a special kind of transaction,

called *control transactions*. There are two types of control transactions. A control transaction of type 1 claims that a site is nominally up. It can only be initiated by the recovering site when it is ready to change its state from down to recovering. It updates the session vector of all operational sites with the recovering site's new session number and obtains a copy of the session vector for the recovering site. A control transaction of type 2 claims that one or more sites are down. This claim is usually made when a site attempts and fails to access a data item on another site. Any site can initiate this type of transaction as long as it is sure that the sites being claimed down are actually down. A transaction of this type reads a copy (likely the local copy) of the nominal session vector and writes 0 to all available copies of the nominal session numbers that correspond to the sites to be claimed down. Control transactions, like all other transactions, follow the concurrency control protocol and the commit protocol used by the DDBS.

These concepts are also used in the section on fail locks. The implementation of these ideas is discussed in [5].

3. Fail Locks

IN THE PROTOCOLS for replicated copy control during failures, the system must keep track of copies that have missed updates due to a failure. If a copy is written while another one is down, the site of the updated copy must do the tracking. This requires knowing about a failure in the middle of the transaction's execution. This automatically requires an extra round of information exchange among sites to know who got the update and who did not. Since this was not done in examples given in the introduction, problems of nonserializability and lost update occur.

We suggest the notion of a fail lock for this task. This idea is adopted from the concept of a lock in concurrency control algorithms where a lock on a data item represents the fact to all other transactions that the locked item is being used by a transaction. In a similar manner, a fail lock is used in a replicated copy control algorithm to represent the fact that a copy of the data item is being updated while some other copies are unavailable due to site failure or network partitioning. For example, when an operational site knows that a particular site is down while updating a data item that has a copy at the down site, it sets a fail lock on behalf of that site. When the failed site is recovering, it collects the fail locks that were set during its failure and marks its copies of the fail-locked items as unreadable.

The basic concept of fail lock is summarized as follows.

Set Fail Lock Operation: A fail lock is set on the copy that is being updated while another copy is not available for update due to a failure. A separate fail lock is set for each known failed site. Thus, if Sites 1 and 3 are down, Site 2 will have two separate fail locks on its copy. Note that the information about another copy not being available due to a failure is present in the session vector (sites for which the session vector has an entry of 0).

Semantics of Fail Lock: Items that are fail locked can be read and written by all

transactions on operational sites. Fail locks can be set repeatedly by incoming updates as long as another site is perceived to be down, but only one fail lock per site is maintained on a copy.

Note that this semantics is different from that used in concurrency control where a locked item cannot be accessed by other transactions unless all transactions were doing reads only.

Release Fail Lock Operation: A fail lock on the copy that was updated is released when an out-of-date copy on a recovering site has been identified and marked. All fail locks for a site are released when a site has marked all data items that are inconsistent with the operational sites. No fail locks exist if all sites are operational.

Note that the transmission of the latest values for a copy from the operational site to the recovering site can be done using the copier transaction [1] or using an update transaction that writes on both copies. The fail lock can also be released as part of the copier transaction or the update transaction.

Now we outline the procedure for processing transactions at the operational sites and the recovering process at the failed site.

3.1. Transaction Processing at the Operational Sites

If all sites are operational, that is, there are no entries in any session vector with a value of zero, all updates on the replicated copies will be successful, subject to the concurrency control. Except for reading the session vector, there is no overhead for the implementation of the above concepts.

There are two implementation choices for posting the updates when no new failures occur, and they both require the participation of the operational sites in two rounds. Note that, if a site has a zero entry in the session vector at the start of a transaction, fail locks are set on the updated copy for the operational sites but no update is sent to the failed copy. In the first choice the updates are sent to all operational sites and in the next round they are committed if no new failures occur. During the first round the other sites can keep the update in a working buffer. In case of further failure, either the second round requests the discarding of the updates sent in the first round or no messages are sent and the updates are discarded based on a time-out mechanism. The second choice involves, first, a round of verification that all operational sites will be able to post the update, and then during the second round the update is sent and committed. This works in the same manner as the two-phase commit protocol [13]. Note that the fail locks for the known failed sites are included with the updates. If a physical write on a copy for the transaction is rejected due to a new failure, a type 2 control transaction is initiated by the processing site and the update for the transaction is aborted.

This protocol can be extended such that a transaction is not aborted due to a new failure. Since fail locks must be set for each failed site along with the update, new failures will generate new fail locks and the process cannot terminate unless there are two consecutive rounds when no new failures occur. In a sense, the updates are

committed in the last round and all previous rounds will be spent in determining the set of failed sites and the fail locks for them. It is a question for the application and the implementation to determine the choice between aborting the transaction or continuing with further rounds.

The responsibility for using the fail lock and releasing it is up to the recovering sites.

3.2. Recovery Process at the Failed Site

As the first step toward becoming operational, the failed site announces that it is ready to process transactions by issuing the control transaction type 1 and thereby updating the session vector of other operational sites with its new session number [5]. The actual session number "$as[k]$" (k being the site identification—ID) of the recovering site still has a value of zero. Next the recovering site must identify the data items that are available to transactions arriving at this site and mark the others as out of date. This marking process can be included in the control transaction type 1. Various cases are discussed in the following paragraph.

If the database is partially replicated, it is possible that the recovering site has the only copy of a data item. Obviously this copy is current and hence is made available to be accessed by any transaction. Next the recovering site collects the set of data items that are fail locked on each operational site and marks them as unreadable on its own site. If a copy of a data item is not available due to the failure of some other site (say site X), it is also unavailable for reading and is marked as before. However, any transaction can write on these marked data items and set the fail locks for site X. However, if a site with the only copy for an item fails, such an item will become unavailable and no recovery for this data item will be necessary. Consider the case when there are k sites with copies of a data item and the k-th site goes down while the other (k-1) sites are already down. If the k-th site recovers before any of the other sites (and it can determine this fact), its copy is immediately available. However, if any of the other sites recover, they will assume that the k-th copy may have been updated and hence the copy is not available for reading but as in the previous case will be marked. Once again these copies are available for writing. The data items on the recovering site that are not marked are considered current and not updated while the site was down. These data items can be read and written by all transactions arriving on the site.

On the other hand if the database is fully replicated, all the fail locks can be obtained from any site.

It is important to note that, during the interval between the reading of fail locks on operational sites and the marking of them on the recovering site, no new fail locks should be set on the operational sites. Essentially the fail locks during recovery should be frozen on the operational sites. This is possible by setting up a buffer for accepting updates arriving at the recovering site during this interval. This buffer should be in stable storage to avoid losing this

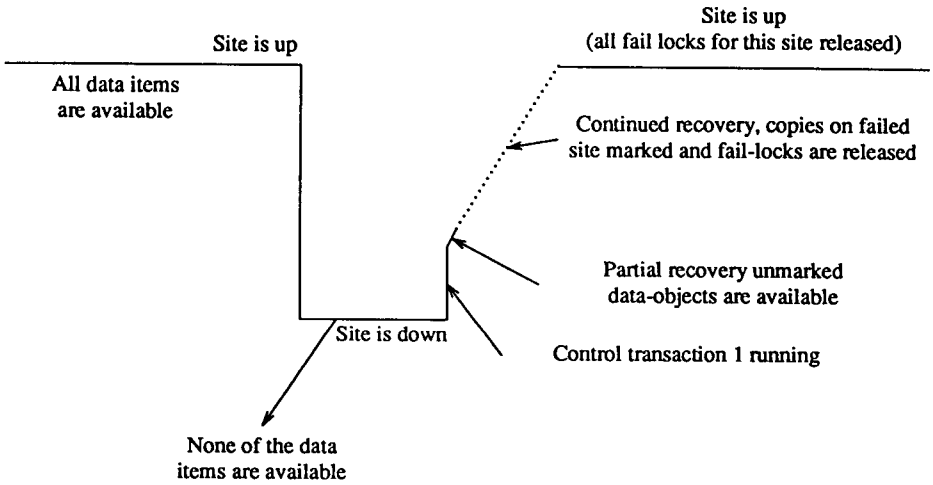


Figure 1. States in Site Recovery and Availability of Data Items for Transaction Processing

information in case of a crash during the recovery.

After the fail locks have been used to mark the copies on the recovering site, the actual session number "$as[k]$" is updated with the new session number. Note that the operational sites can release the fail locks for this site as soon as the recovering site has collected all the fail locks. The updates from the buffer are posted and the recovery is almost complete. All that is left for complete recovery is to get the marked data items renovated. Such items can be updated and the marks removed by future user transactions or by forcing a copier transaction that reads all marked data items and writes on them. An immediate copier transaction will also provide robustness against failures of operational sites.

Figure 1 illustrates the process of site recovery and availability on data items. The site recovery procedure is summarized as follows.

1. When a site k starts, it loads its actual session number $as[k]$ with 0, meaning that the site is ready to process control transactions but not user transactions.

2. Next, the site initiates a control transaction of type 1. It reads an available copy of the nominal session vector and refreshes its own copy. Next this control transaction writes a newly chosen session number into $ns_i[k]$ for all operational sites i including itself, but not $as[k]$ as yet.

3. Using the fail locks on the operational site, the recovering site marks the data copies that have missed updates since the site failed. Note that steps 2 and 3 can be combined.

4. If the control transaction in step 2 commits, the site is nominally up. The site converts its state from recovering to operational by loading the new session number into $as[k]$. If step 2 fails due to a crash of another site, the recovering site must initiate a control transaction of type 2 to exclude the newly crashed site, and then must try step 2 and 3 again. Note that the recovery procedure is delayed by the failure of another site, but the algorithm is robust as long as there is at least one operational site coordinating the transaction in the system.

The proof of correctness of this procedure is in [5].

3.3. Setting Fail Locks on All Replicated Copies

Since an update on a data item will write on all operational copies, a natural choice is to set fail locks on all updated copies. This choice has one advantage and one disadvantage. The disadvantage is that the recovering site must go to all sites since several updates may be in progress, and the recovering site may find itself collecting duplicate information about the fail lock on each data item. In addition, the release of fail locks must access all copies.

On the other hand, if multiple sites fail concurrently even if a single site is operational, the rest of the system can be recovered by using the fail locks. However, data items for which all copies are simultaneously down cannot be recovered unless we can store the fail locks on stable storage and identify and wait for the last site to recover that went down with a good copy.

3.4. Comments on the Implementation of Fail Locks

A site failure server has been implemented in the experimental system RAID [4]. RAID is a prototype distributed database system and is a collection of autonomous servers connected by a communication system. Each site runs servers that implement access management with stable storage, transaction execution, concurrency control, recovery management, and other services for transaction processing.

The communication system can run either UDP/IP (user datagram protocol/internet protocol) datagrams or a high-speed ethernet protocol that we are building. High-level calls exist for services such as reliable multicast necessary for distributed transaction commitment. A name server has been implemented to provide location dependent addressing for various servers. This server keeps track of failure/recovery of sites and the integration of new sites to the system. The communication system automatically caches the addresses of all servers and contacts the name server only when the cache must be refreshed.

We have learned the following details about implementations in this work.

After a recovering site has issued a control transaction of type 1, it has informed the other sites that it is ready to accept the updates from them. However, while the recovering site is collecting fail locks, its actual session number should be assigned a value of zero. Only after a site has collected the fail locks successfully should it load its actual session number with the value saved in the stable storage. As discussed earlier, the updates received during recovery should be saved on stable storage in a buffer. This protects the recovery process from failure of other sites [5]. Note that a crash during recovery will only postpone the recovery procedure.

In implementation, it may not be feasible to place the fail lock bits as part of the data item structure. For example the following obvious choice may not be feasible

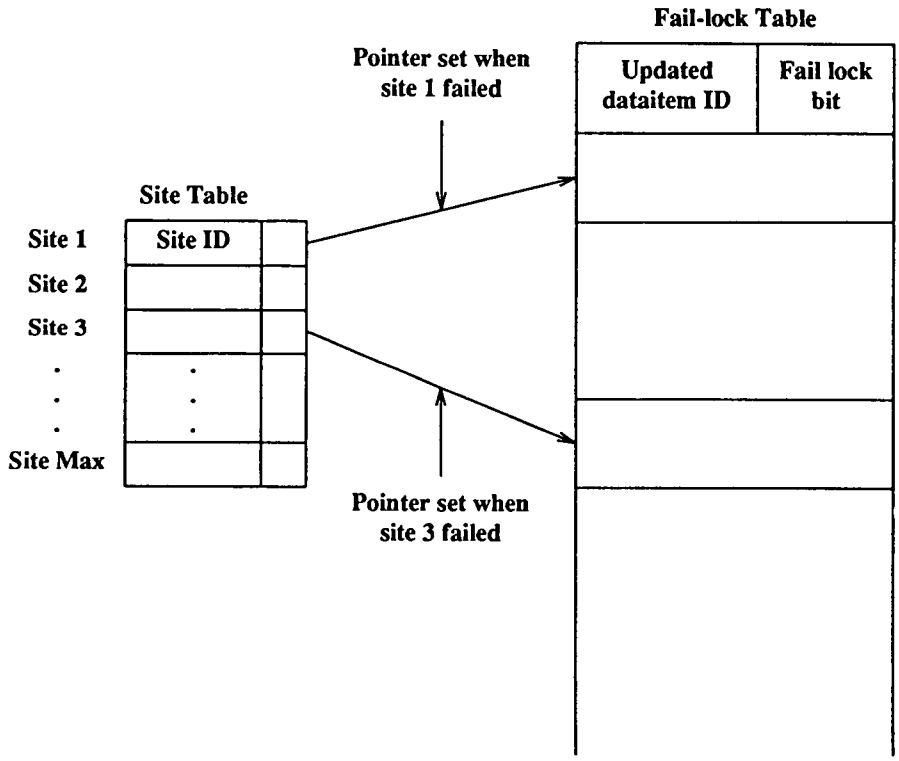


Figure 2. Implementation of Fail Lock Table on Each Operational Site

because this information is needed in memory for processing.

data item	fail lock with failed site ID

The next choice is to implement a fail lock table similar to a lock table that is kept in memory for concurrency control. We need three pieces of information in this table.

(a) Fail lock bit

(b) Site-ID bit

(c) Time stamp when the fail lock was set

A possible implementation that we are experimenting with is shown in Figure 2.

The "fail lock table" entries are made when one or more sites are down. This table grows as updates are posted. Each update creates an entry in this table. There is a "site table" that contains a pointer to the fail lock table. This pointer is set when a site fails. When a site recovers, all fail lock table entries from its pointer to the last entry in the table or up to the pointer for another site are considered as fail locked by this site only and deleted assuming there is no failed site pointing to an earlier entry. When all sites recover, the fail lock table entry is empty. The only disadvantage of this structure is the redundant entries in the fail lock table. But such replication can be processed along with the redundant fail lock information from other sites.

3.5. Missing Lists

Another practical implementation [5] is to use a *missing list* (ML). Conceptually, a missing list is a two-dimensional array *ML*: $\{item\} \times \{site\} \rightarrow \{1, 0\}$, where $ML[X, i] = 1$ means x_i has missed updates. In order to save storage space, an ML can be implemented in various ways, for example, as a list of pairs (X, i) for non-zero elements in the ML. The elements of the ML can be seen as data items that are augmented to the database, but need be stored in volatile storage only. Access to elements should be under concurrency control. Each site maintains an ML. Unlike *NS*, MLs at different sites are considered as different data items, rather than copies of the same logical data. A pair (X, k) in ML at site i means that $x_i \in X$, $x_k \in X$, and x_k has missed an update which is done to x_i. Our algorithm can work with MLs as follows. A write operation WRITE(X) writes to x_i for all $i \in X$ such that site i is nominally up. It removes (X, i), if any, from the MLs at the sites to which it writes a copy of X successfully, and it adds (X, j) into these MLs for all j such that $x_j \in X$ and site j is not available for the transaction. When site i is recovering, it looks up the MLs at all operational sites. If (X, i) appears in an ML, site i removes the entry (X, i) from all MLs of nominally operational sites and marks its own copy x_i as unreadable. Site i also forms its own ML using the entries (X, j), $i \neq j$, seen in the MLs at other operational sites. It should be noted that, under this mechanism, as along as a site has an up-to-date copy of a data item, the ML of this site has the precise information on where the copies of the data item have missed updates.

3.6. Network Partition Merging

The ideas presented in the previous sections deal with the problem of failed site integration with the operational sites. We believe that the solution to the site failure problem and the concept of nominal session numbers are applicable to the merging of network partitions. Full details have not been worked out but the direction of research is outlined as follows.

The distinction between the problems of network partition and site failure is clear. In a site failure problem, the operational sites in the system can assume that no activity occurs at the failed site. Thus the failed site needs to integrate with the rest of the system and obtain updates missed during its failure. This means that integration is required only in one direction (from the failed site to the operational sites). In a network partition problem, the system may allow updates on different data items in different partitions. For example, updates can be allowed on data items holding true-copy tokens [10]. When two partitions merge, each partition needs to obtain missed updates from the other partition. This can be accomplished by integrating the sites of a partition one by one with the other partition. When a site obtains all updates from another partition, it is considered integrated in one direction. A site is fully integrated with another partition if integration in both directions has been completed. Two partitions are fully

integrated when all sites in each partition have fully integrated. Integration in either direction follows a protocol similar to a failed site recovery protocol discussed in this paper. The granularity at which the integration takes place is up to the implementation.

4. View Serializability for Processing in Network Partitioning

SERIALIZABILITY HAS BEEN USED as a criterion for correct concurrency control [11]. Basically a conflict graph that contains the transactions as nodes and read-write or write-write conflicts as edges is created. If this graph is acyclic, the transactions are serializable; otherwise, not.

When a network partition occurs, if processing is allowed in both partitions, only acyclic partial conflict graphs are allowed in each partition. However, when merged, a global conflict graph may contain a cycle and hence the processing in each partition must be further restricted. A survey [6] contains various techniques to deal with network partitions.

In applications such as banks, airline reservations, and battle management systems, there are many user transactions that like to just read the database values. In other words, the users like to get a view of a correct database state. In network partition environment, even if the current view is not available, an earlier version may be acceptable to users. For example, if I call a bank to find the balance in my account, the following answer may be acceptable: your balance is this amount; however, some checks may not have been processed. The cause of unprocessed checks may be delay due to a failure of some part of the system or network partition. Of course, when I actually go to withdraw the funds, my transaction becomes an update and could be rejected. Another example is a situation in which one calls an airline to check if certain seats are available versus the situation when one actually makes reservations.

This leads us into a correctness criterion called "view serializability" for concurrency control where read-only transactions are treated differently from the update transaction. We have learned of this notion from work on update serializability in locking [14]. It may have similar features as the notion of weak-serializability discussed in the literature.

The correctness criterion based on *view serializability* requires the following two conditions:

1. The update transactions do not create a cycle in the conflict graph, and
2. the *read-only* transactions considered *one at a time* in the conflict graph do not create a cycle.

This notion has been used for locking based concurrency control in hierarchically structured database systems [14]. Interestingly, this notion also contributes toward freedom in allowing read-only transactions during network partitions. The basic idea of a protocol based on this notion is as stated in sections 4.1 and 4.2.

4.1. Update Transactions

During network partitioning, update transactions have to be restricted so that the global conflict graph does not produce a cycle. For example, a token can be associated with each data item [10] to achieve this. This token is resident on, at most, one copy, for example, the copy where the last update took place.

All data items needed by an update transaction have to be in the partition in which the transaction wants to execute. Thus, no data item can be updated in two partitions, causing a conflict. This will ensure condition (1) for serializability.

4.2. Read-Only Transactions

A read-only transaction is allowed in each partition as long as it does not create a cycle with the update transactions in that partition. This processing does not require information about what is going on in the other partition and hence can go on freely.

We illustrate the above protocol via the following example.

Let there be two data items X and Y. Assume they are fully replicated on two sites A and B. At the time of partition, let site A contain the token for X and let site B contain the token for Y.

Let there be five update transactions T_1, T_2, T_3, T_4, and T_5. Let T_1 and T_4 execute on site A, T_3 and T_5 execute on site B, and T_2 execute on both sites.

The versions of X due to updates are represented as X', X'', X''', and so on, and the versions of Y are represented as Y', Y'', Y''', and so on. The processing of transaction proceeds as follows:

	Site A	. . .connected. . .	Site B
Data items (Initial versions)	X Y		X Y
T_1 (updates X)	X' Y		X' Y
T_2 (updates X and Y)	X'' Y'		X'' Y'
T_3 (updates Y)	X'' Y''		X'' Y''
	--. . .Network Partition. . .--		
	(site A has token for X and site B has token for Y)		
T_4 (arrives on site A and updates Y)	X''' Y''		
T_5 (arrives on site B and updates Y)			X'' Y'''
T_6			X'' Y'''

(arrives on site B and reads X and Y) T_7 (arrives on site A and reads X and Y)

X''' Y'' (unchanged)

(unchanged)

--. . .Communication Established. . .--
(Network Merges)

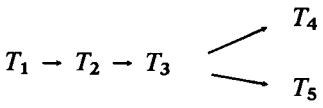
At the time of network partition, the conflict graph on each site is as follows

$$T_1 \rightarrow T_2 \rightarrow T_3$$

After the processing of T_4 and T_5, the conflict graph on each site is as follows

Site A: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4$ Site B: $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_5$

The global conflict graph is



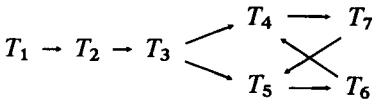
Since T_4 and T_5 cannot conflict, there are no edges between T_4 and T_5.

Since T_6 is a read-only transaction, the conflict graph on site B contains an edge $T_5 \rightarrow T_6$.

Similarily T_7 is a read-only transaction causing an edge $T_4 \rightarrow T_7$. Obviously there cannot be a conflict due to these edges since there are no edges going back from T_6 and T_7.

In the global conflict graph, after the partition merges, we must include edges $T_6 \rightarrow T_4$, since T_6 read X before T_4.

Similarly the edge $T_7 \rightarrow T_5$ exists since T_7 read Y before T_5. These edges are known only after the partition merges. The global conflict graph after the establishment of communication is as follows:



Obviously there is a cycle in this graph and in the protocol based on conventional serializability. However, if only T_7 or T_6 is considered along with T_1 to T_5, the conflict graph is acyclic. Hence the conflict graph satisfies the conditions for view serializability. Essentially what it means is that T_7 sees the view produced by the

history $T_1T_2T_3T_4$, and T_6 sees the view produced by the history $T_1T_2T_3T_5$. Since both $T_1T_2T_3T_4$ and $T_1T_2T_3T_5$ are serializable independently and jointly, they both see a consistent state of the database even though it may not be current.

4.3. Assertion

If a read-only transaction does not create a cycle with the partial conflict graph during partition, it will not create a cycle in the global graph involving update transactions from all partitions.

Proof: Let $T_1, \ldots, T_n$ be the transactions processed before the partition occurs. Obviously transactions $T_1, \ldots, T_n$ are serializable and their graph does not contain a cycle. Let T_{u1} and T_{r1} be the update and the read-only transactions that arrive in partition 1, and T_{u2} and T_{r2} be the update and read-only transactions that arrive in partition 2. For T_{r1} to be successful, it cannot have a cycle with $T_1, \ldots, T_n$. Let us assume it serializes after T_n. Since T_{r1} cannot see the update of T_{u2}, in case of a conflict, it must precede it in any serialization order. Thus an edge $T_{r1} \rightarrow T_{u2}$ is possible in the global conflict graph but not vice versa. In addition since T_{u1} and T_{u2} are allowed during the partition (say using the tokens) only if they do not conflict, there are no edges between them. Thus there is no edge/path starting from any of the updates (T_{u2}) occurring in partition 2 to any of the read-only transactions (T_{r1}) either directly or via any of the updates (T_{u1}) occurring in partition 1. Hence T_{r1} cannot have a cycle involving T_{u1} and T_{u2}. Thus, T_{u1}, T_{u2}, and T_{r1} cannot create a cycle in the global conflict graph.

In this proof, we have considered only the cycles that can occur during the partition and this is sufficient. The other transactions are serializable due to the correct concurrency control and do not have incident edges coming from the transactions processed after the partition.

5. Conclusions

FROM THIS STUDY, we find that, although single failures may be easy to handle, multiple site and network partition failures require further research. Increasing of the availability of the database implies additional bookkeeping and requires several new notions such as session numbers, fail locks, and view serializability. The real problem with a distributed database system is that no site can be fully sure about the status of another site, but at the same time it would like to continue processing transactions. This requires the notion of nominal view of the status of all sites and leads us into nominal session vectors. The implementation, maintenance, and utility of such data structures and notions is yet to be fully explored. An experimental study [3] reports overheads due to fail lock maintenance, control transactions, and copier transactions. It also discusses the data availability on a recovering site.

We find that these are real problems. For example, in an army, platoons and

companies within a division may be isolated or captured. The decision making must go on and whoever can communicate with the commander must provide him with status information which must be consistent even if not current. The status of other platoons or companies may be perceived but the operations must be done autonomously and bookkeeping should be carried out for others so that they can integrate or recover to a consistent state.

Another example is a computer science or business department of a college where faculty are equivalent to sites and students act as transactions. Though each student needs the same four faculty on his committee to be around from the time he/she starts the Ph.D. and finishes it, the status (leaves, sabbaticals, change of jobs, denial of promotions, not around) of a faculty is difficult to predict. Students use nominal session numbers and are always asking other faculty about the status of other faculty and at the same time continuing their progress toward their Ph.D. Faculty members use fail locks to update the absent faculty members about the events that happened when they were away. The chairman of the department likes to get a correct view of the status of faculty and students and can use view serializability. I do not know of any student who did not finish his/her Ph.D. because the status of faculty in his/her committee was changing or even because a committee member left the university.

REFERENCES

1. Attar, R.; Bernstein, P. A.; and Goodman, N. Site initialization, recovery, and backup in a distributed database system. *IEEE Transactions in Software Engineering*, SE-10, 6 (November 1984), 645–650.

2. Bernstein, P. A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases. *ACM Transactions in Database Systems*, 9, 4 (December 1984), 596–615.

3. Bhargava, B., and Noll, P. An experimental analysis of replicated copy control during site failure and recovery. Technical report, CSD-TR-692. Purdue University, May 1987.

4. Bhargava, B., and Riedl, J. The design of an adaptable distributed system. *IEEE COMP-SAC Conference* (October 1986), 114–122. (Current version available as Purdue University technical report CSD-TR-691, July 1987.)

5. Bhargava, B., and Ruan, Z. Site recovery in replicated distributed database systems. *Sixth IEEE International Conference on Distributed Computing Systems* (May 1986), 621–627.

6. Davidson, S. Consistency in partitioned networks. *ACM Computing Surveys*, 17, 3 (September 1985), 341–370.

7. Gifford, D. Weighted voting for replicated data. *Proceedings of the Seventh ACM Symposium on Operating System Principles* (December 1979), 150–161.

8. Hammer, M. M., and Shipman, D. W. Reliability mechanism for SDD-1: A system for distributed databases. *ACM Transactions in Database Systems*, 5, 4 (December 1980), 431–466.

9. Lampson, B., and Sturgis, H. Crash recovery in a distributed data storage system. Xerox PARC report. Palo Alto, CA, April 1976.

10. Minoura, T., and Wiederhold, G. Resilient extended true-copy token scheme for a distributed database system. *IEEE Transactions in Software Engineering*, SE-8, 3 (May 1982), 173–188.

11. Papadimitrou, C. H. Serializability of concurrent updates. *JACM*, 26 (October 1979),

631–653.

12. Skeen, D. Nonblocking commit protocols. *Proceedings of the 1981 ACM-SIGMOD Conference on Management of Data*, New York (1981), 133–147.

13. Skeen, D., and Stonebraker, M. A formal model of crash recovery in distributed system. *IEEE Transactions in Software Engineering*, SE-9, 3 (May 1983), 219–227.

14. Yannakakis, M. Serializability by locking. *JACM*, 31 (April 1984), 227–244.